Assignment 2

CSE2215 Software Engineering Methods
2021/2022

Group 11b
Aleksandra Andrasz
Ciprian Stanciu
Matyáš Pokorný
Mike Raave
Milan de Koning

Tools

We have selected CodeMR as a tool to compute our code metrics.

Class-wide refactoring

Metrics

We have chosen to select the classes that require refactoring work according to their (lack of) cohesiveness and coupling. The aim is to have a good balance between cohesion, as high cohesion points to a simpler design complexity and leads to reusable code that is easy to understand, and coupling, as low coupling simplifies modification. The metrics we will take into account are:

- LCOM (Lack of Cohesion of Methods): this measures the cohesiveness of the class methods. A high score on this metric means that this class is a problem and it should lead to splitting the class into smaller classes that only have one responsibility. This is based on the number of attributes shared between the class's methods.
- LCAM (Lack of Cohesion among Methods): this measures the cohesiveness of the class methods. A high score on this metric means that this class is a problem and it should lead to splitting the class into smaller classes that only have one responsibility. This measure is based on the amount of shared parameter types between class methods.
- LTCC (Lack of Tight Class Cohesion): this measures the level of cohesion between the public methods of the class (how often they work with the same fields). A high score on this metric should lead to splitting the class into smaller classes that only have one responsibility as well. This measure is based on the amount of class fields that are accessed from different public methods.
- CBO (Coupling Between Objects): this relates to the number of classes a class is coupled to. When this metric is high, care should be taken to ensure that the classes are coupled as loosely as possible.

Threshold

We used the thresholds provided by CodeMR (low, low-medium, medium, medium-high, high, very-high).
We have chosen to improve classes that score at least medium-high in two or more of our chosen metrics.

Details

Before:	LCOM	LCAM	LTCC	CBO
`RoomsServiceImpl*`	0.708	0.821	0.220	31
`ApiDateTime`	0.167	0.808	0.636	4
`ReservationServiceImpl**`	0.660	0.751	0.214	24
`GroupServiceImpl***`	0.688	0.514	0.619	12
`ApiDate`	0.617	0.783	0.771	2

*from room microservice
**from reservation microservice
***from authentication microservice

After:	LCOM	LCAM	LTCC	CBO
`RoomsServiceImpl`	0.657	0.79	0	26
`ApiDateTime`	0.167	0.7	0.286	3
`ApiDateTimeUtils`	0	0.333	0	4
`ReservationServiceImpl`	0.417	0.694	0.143	12
`GroupServiceImpl`	0.533	0.375	0.200	9
`ApiDate`	0.600	0.540	0.470	0
`ApiDateUtils`	0.000	0.500	0.000	3

`RoomsServiceImpl`

We refactored the `RoomsServiceImpl` class by splitting it across domain borders. We extracted all methods related to handling faults and moved them into their own service class (`FaultServiceImpl`). This change drastically reduced the number of methods in the class and helped eliminate one the injected dependencies. Overall this helped our code cohesion.

`ApiDate`

We improved the `ApiDate` by first changing to a consistent type for number variables, as we were using shorts, ints and longs interchangeably. The class now only uses integers. We

also simplified the storage of the date a bit, by storing a day, month and year instead of just a day and year. This made the code a lot more straightforward and intuitive. The biggest refactoring operation that improved the metrics the most was **Extract class** refactoring. We moved all methods that were not responsible for storing or retrieving the data into a new class, called `ApiDateUtils`. These methods included: A method that returns the current day, A method that parses a `String` into a new `ApiDate` and a compare method. After this refactoring operation, the `ApiDate` class mostly contains methods that store, retrieve and validate information about dates, while the `ApiDateUtils` class is responsible for methods that perform operations on dates. As you can see, this improved all metrics. (except CBO which went from 2 to 3, which is still nowhere near problematic.) Especially LCAM and LTCC saw great improvement, since the responsibilities of the `ApiDate` class have now been separated nicely.

`ApiDateTime`

`ApiDateTime` was reported by CodeMR as being a problematic class, with a high lack of cohesion. Both LCAM and LTCC scores had fairly high values, with LCAM being particularly problematic. This pointed to the class using needlessly many types of attributes in its methods. In this case, as the class is basically implementing functionality for a pair of `ApiDate` and `ApiTime`, there was also a constructor for all of the components of the two smaller classes alongside a constructor with the two base classes. This is unnecessary and harmful, as type changes in one of `ApiDate` or `ApiTime` would lead to changes in `ApiDateTime`'s constructor, so we removed that constructor and fixed all parts of the code where it was referenced. This led to the LCAM metric by quite a bit. Another change was moving the static methods of the class into a separate Utils class, as there is no need to have them in the main entity class, and keeping them there makes the class have more than one role. These methods were for parsing other types of date and time storage to our entity. Although the LCAM metric did not go down by a lot in absolute terms, it was enough for CodeMR to remove it from its list of problematic classes. However, the LTCC metric was greatly improved. A further improvement would be hard to realize, as most methods override or implement basic functionality that must always be there, and bring the score up since the class itself is quite small. The threshold provided by CodeMR for cohesion changed from ‘high’ to ‘low-medium’.

`GroupServiceImpl`

We improved the `GroupServiceImpl` class by first changing some of the input variable types to be more consistent to the rest of the methods, since instead of a Group object, the groupId could also be provided, which was more inline with how the rest of the methods were set up. The amount of coupling was decreased by using one exception class where we were first using 3 different ones, quite interchangeably. So, the 2 least used in the project are deleted and replaced with the most used exception. This changed the CBO metrics from 12 to 9 and the general coupling metrics from medium-high to low-medium.
Lastly, the `getGroupsOfCurrentUser` method was deleted, because we realized that the `getGroupsOfUser` method was essentially doing the same thing, but in a different way.

This change separated the functionality between the classes well, since the `getGroupsOfUser` and `getGroupsOfSecretary` methods are now only used separately from each other. These changes did seem to make a large impact on mostly the LCOM and LTCC metrics (as can be seen in the tables above), since the type of input variables is more consistent and the functionality is a bit more separate from each other, which also impacted the LCAM metric in a positive way.

ReservationServiceImpl

We decided to improve ReservationServiceImpl as CodeMR identified it as a problematic class. This class had high scores in coupling and lack of cohesion which was reflected in the high score of CBO and LCAM metric. Firstly we identified which logic could be moved to different classes using Move Method refactoring in order to reduce coupling. The verification if the maintenance is ongoing was moved to Closure. The operation of converting Timestamps to ApiDateTime and the other way was moved to ApiDateTime. Converting ReservationRequestModel to Reservation was moved to Reservation entity. All those changes lowered the coupling of ReservationServiceImpl since the class relied less on other classes. To improve cohesion and further reduce coupling, two new classes were introduced following the Extract Class refactoring method. One of the classes was UserValidationService which takes care of checks regarding users and groups, this meant that ReservationSericeImpl does not use classes such as UserModel, GroupService or UserService and thus lowered coupling. Similarly second class, RoomValidationService, performs checks regarding rooms and buildings which also removed some classes that ReservationServiceImpl relied upon previously. It also meant that the functionality of refactored class was more cohesive as it only took care of operations connected to creating and modifying reservations leaving all the checks for the new classes. The improvements were reflected in code metrics as we observed that each of the metrics we chose was lowered, and the class is no longer considered problematic by CodeMR.

Method-wide refactoring

Metrics

We have chosen to select methods that require refactoring based on their complexity and number of method calls. The exact metrics provided by CodeMR that we used are:

- MCC (McCabe Cyclomatic Complexity). A high score on this metric means that the method is probably quite complex, since there exist a lot of independent paths through the method. This can be fixed by either moving code to new methods or eliminating nesting inside the method.
- #MC (Number of method calls). A high score on this metric means that the method calls a lot of other methods and probably means that the method is highly coupled. A good way of refactoring this and improving this method is to split up the method in smaller chunks, or remove duplicate method calls (such as getters).

Threshold

For the method metrics, we chose to improve methods that have at least 5 for MCC and at least 10 for #MC. We think that using these metrics give the best insight into the quality of the methods.

Details

Before:	MCC	#MC
`RoomServiceImpl::setupChain`	12	19
`ReservationsServiceImpl::editReservation`	10	32
`RoomServiceImpl: addRoom`	5	21
`RoomServiceImpl: addEquipment`	5	16
`ReservationServiceImpl::validateRoom`	8	15

After:	MCC	#MC
`RoomServiceImpl::setupChain`	1	8
`ReservationsServiceImpl::editReservation`	3	17
`RoomServiceImpl::addRoom`	1	5
`RoomServiceImpl::addEquipment`	3	6
`ReservationServiceImpl::validateRoom`	2	11

`RoomServiceImpl::setupChain`

One of the worst offenders was `RoomServiceImpl::setupChain`. This method used to have multiple `if`-statements that handled all the filters and all the option parsing. We extracted all the option handling code, split it and moved it to its respective filter classes. This not only improved the metrics significantly, but also better respected the principle of encapsulation of OOP.

`ReservationsServiceImpl::editReservation`

Before refactoring, the `editReservation` method was quite complicated and coupled with other methods. This was visible in high cyclomatic complexity with MCC score equal to 12 and the number of calls to methods equal to 32. Most of the refactoring was done with the Extract Method refactoring technique to reduce both complexity and coupling.
The logic for verifying if a person has permission to modify the reservation was put in `UserValidationService::userCanModifyReservation`. The logic for checking for the existence of rooms was moved to the method `RoomValidationService::validateRoom`. This was connected to refactoring of the class ReservationServiceImpl.
Extract Method strategy was used to create methods `ReservationsServiceImpl::checkReservationExists` and `ReservationsServiceImpl::validateRequest`. The long list of parameters was replaced with ReservationRequestField as all the attributes were also fields of that object and make sense with method functionality.

RoomServiceImpl::addEquipment

For the `RoomServiceImpl::addEquipment` method, the biggest problem was the large number of method calls. This is solved by using the Extract Method strategy to split the functionality between 3 methods, where we were first using 2 methods, of which the addEquipment method was the largest. The extraction of an additional method also slightly reduced the cyclomatic complexity of the method, but increased it for the two sub methods (`addEquipmentToRoom` and `addEquipmentToSystem`) and made the complexity of the methods more balanced. Many of the method calls were calls which could only be made on that line in the method, so that made it difficult to extract variables to reduce the method calls.
Right now the method calls are divided between 3 methods, but the number of method calls is also reduced overall, since the structure of the methods is also more optimized.

RoomServiceImpl::addRoom

We refactored the `addRoom` method using the Extract method strategy, moving the code that converted a `Room` class into a `RoomModel` and the other way around into the `Room` class. This greatly reduced the amount of method calls, because the fields from `Room` could now be accessed from the class itself, reducing the total amount of method calls. The method to convert it the other way around still required method calls, but all of those are now in a method dedicated to only converting the class. All of the code that found the building that the room was supposed to be in and the complexity that went with that was also moved into a `getBuilding` method, reducing the amount of method calls and complexity as well.

ReservationsServiceImpl::validateRoom

For the `ReservationServiceImpl::validateRoom` method, there were quite a lot of method calls, with a moderate complexity. In order to decrease the complexity of the method and to lower the amount of method calls, we refactored a conditional method to reduce the number of code paths and then we extracted the method into two, the other being `validateBusinessHours` which takes care of validating whether a reservation is made during the building business hours. To further reduce the number of method calls, we also added some intermediary variables such as `building` and `closure` which hold the values of certain method call results that are repeated throughout the method.